Pseudocode for Reading and Parsing Data from File:

Procedure ReadDataFromFile(file\_path):

Input: file\_path - The path to the input file containing course data

Output: data\_structure - The data structure to store course objects

// Initialize an empty data structure

data\_structure = CreateEmptyDataStructure() // Replace CreateEmptyDataStructure() with the appropriate data structure creation function

// Open the file for reading

file = open(file\_path, "r")

// Loop through each line in the file

while not end\_of\_file(file):

line = read\_line(file)

// Parse the line to extract course information

course\_info = split\_line(line, ",") // Split the line using comma as the delimiter

// Create a course object with the extracted information

course\_object = CreateCourseObject(course\_info) // Replace CreateCourseObject() with the function to create course objects

// Add the course object to the data structure

data\_structure.add(course\_object) // Replace add() with the appropriate function to add elements to the data structure

// Close the file

close(file)

// Return the populated data structure

return data\_structure

End Procedure

Procedure CreateCourseObject(course\_info):

Input: course\_info - A list containing course information (course\_number, course\_title, prerequisite1, prerequisite2, ...)

Output: course\_object - An object representing a course with its title and prerequisites

// Extract course information from the list

course\_number = course\_info[0]

course\_title = course\_info[1]

prerequisites = course\_info[2:] // Extract all elements from index 2 to the end

// Create a course object

course\_object = CreateObject(course\_number, course\_title, prerequisites) // Replace CreateObject() with the appropriate function to create objects

// Return the course object

return course\_object

End Procedure

Pseudocode for Menu Functionalities:

Procedure DisplayMenu():

Output: Menu options

Print "Menu:"

Print "1. Load Data Structure"

Print "2. Print Course List"

Print "3. Print Course"

Print "4. Exit"

End Procedure

Procedure MainMenu():

// Initialize variables

data\_structure\_loaded = False

data\_structure = None

// Loop until the user chooses to exit

Repeat:

DisplayMenu()

choice = read\_integer\_input()

// Process user's choice

If choice == 1:

data\_structure = ReadDataFromFile("input\_file.txt") // Replace "input\_file.txt" with the actual file path

data\_structure\_loaded = True

Print "Data structure loaded successfully."

ElseIf choice == 2:

If data\_structure\_loaded:

PrintCourseList(data\_structure)

Else:

Print "Data structure is not loaded. Please choose option 1 to load data structure."

ElseIf choice == 3:

If data\_structure\_loaded:

course\_number = read\_course\_number\_input() // Replace with a function to read course number from the user

PrintCourse(data\_structure, course\_number)

Else:

Print "Data structure is not loaded. Please choose option 1 to load data structure."

ElseIf choice == 4:

Exit // Exit the program

Else:

Print "Invalid choice. Please try again."

Until choice != 4

End Procedure

Procedure PrintCourseList(data\_structure):

Input: data\_structure - The data structure containing course objects

Output: None

// Sort the course information by alphanumeric course number from lowest to highest

sorted\_data = SortDataByCourseNumber(data\_structure) // Replace with the appropriate sorting function

// Print the sorted list to display

For each course in sorted\_data:

Print course.course\_number, course.course\_title

End Procedure

Procedure PrintCourse(data\_structure, course\_number):

Input: data\_structure - The data structure containing course objects

course\_number - The course number for which details are to be printed

Output: None

// Find the course object with the given course number

course = FindCourseByNumber(data\_structure, course\_number) // Replace with the function to find a course by its number

// If the course is found, print its title and prerequisites

If course is not None:

Print "Course Title:", course.course\_title

If course.prerequisites is not empty:

Print "Prerequisites:", course.prerequisites

Else:

Print "No prerequisites for this course."

Else:

Print "Course not found."

End Procedure

Pseudocode for Alphanumeric Sorting:

Procedure MergeSort(data):

Input: data - The list of course objects to be sorted

Output: sorted\_data - The list of course objects sorted in alphanumeric order

// Base case: If the list has only one element, it is already sorted

If length(data) <= 1:

Return data

// Split the data into two halves

mid = length(data) / 2

left\_half = data[0:mid]

right\_half = data[mid:]

// Recursively sort both halves

left\_sorted = MergeSort(left\_half)

right\_sorted = MergeSort(right\_half)

// Merge the sorted halves

sorted\_data = Merge(left\_sorted, right\_sorted)

// Return the merged and sorted data

Return sorted\_data

End Procedure

Procedure Merge(left, right):

Input: left - The left half of the course objects to be merged

right - The right half of the course objects to be merged

Output: merged - The merged and sorted list of course objects

// Initialize variables

merged = []

left\_idx = 0

right\_idx = 0

// Compare and merge elements from left and right into merged

While left\_idx < length(left) and right\_idx < length(right):

If left[left\_idx].course\_number <= right[right\_idx].course\_number:

Append left[left\_idx] to merged

left\_idx = left\_idx + 1

Else:

Append right[right\_idx] to merged

right\_idx = right\_idx + 1

// Append any remaining elements from left and right

While left\_idx < length(left):

Append left[left\_idx] to merged

left\_idx = left\_idx + 1

While right\_idx < length(right):

Append right[right\_idx] to merged

right\_idx = right\_idx + 1

// Return the merged list

Return merged

End Procedure

Evaluation:

Considering the advantages of each data structure, creating a one-dimensional vector would be a plus for handling the data in this assignment. Since we are only dealing with courses and reading files, using a vector would also result in less memory usage. However, a major disadvantage of vectors is that elements cannot be easily deleted, and they do not support multiple data types.

On the other hand, creating a hash table for the project offers several benefits, including organized and efficient storage. The hash table can be accessed using keys, allowing for easy creation, deletion, and retrieval of elements within the project. This versatility enables the handling of unique elements and ensures synchronization. Nonetheless, the drawback of using hash tables is that they may slow down the process due to synchronization.

Creating a tree data structure provides excellent organization as data can be stored in left or right branches and expanded accordingly. This allows for efficient searches and other operations throughout the project. However, similar to the hash table, modifying a tree can be time-consuming.

Out of all three data structures, I would prefer using hash tables because despite their potential speed impact, they offer numerous advantages. The organization they provide aligns well with the current status of the project, where we primarily want to sort courses. Moreover, if the company decides to add more functionality to the project in the future, hash tables would accommodate different items seamlessly. This flexibility and adaptability are the main reasons why I would lean towards using hash tables.